Parse Tree

PROGRAM

`<p4program> := <declarationList>`

`<declarationList> := { <declaration> | ; /* empty declaration */ }*`

`<declaration> := <variableDeclaration> | <externDeclaration> | <actionDeclaration> |`
`  <parserDeclaration> | <typeDeclaration> | <controlDeclaration> | <instantiation> |`
`  <errorDeclaration> | <matchKindDeclaration> | <functionDeclaration>`

`<nonTypeName> := IDENTIFIER | apply | key | actions | state | entries | type`

`<name> := <nonTypeName> | TYPE_IDENTIFIER`

`<nonTableKwName> := IDENTIFIER | TYPE_IDENTIFIER | apply | state | type`

`<parameterList> := { <parameter> { , <parameter> }* }`

`<parameter> := { <direction> } <typeRef> <name> { = <expression> }`

`<direction> := in | out | inout`

`<packageTypeDeclaration> := package <name> <optTypeParameters> ( <parameterList> )`

`<instantiation> := <typeRef> ( <argumentList> ) <name> ;`

`<optConstructorParameters> := { ( <parameterList> ) }`

PARSER

`<parserDeclaration> := <parserTypeDeclaration> <optConstructorParameters>`
`  { <parserLocalElements> <parserStates> }`

`<parserLocalElements> := { <parserLocalElement> }*`

`<parserLocalElement> := <variableDeclaration> |  <instantiation> /* | <valueSetDeclaration> */`

`<parserTypeDeclaration> := parser <name> <optTypeParameters> ( <parameterList> )`

`<parserStates> := <parserState> { <parserState> }*`

`<parserState> := state <name> { <parserStatements> <transitionStatement> }`

`<parserStatements> := { <parserStatement> }*`

`<parserStatement> := <assignmentOrMethodCallStatement> | <directApplication> | <parserBlockStatement> |`
`  <variableDeclaration> | <emptyStatement>`

`<parserBlockStatement> := { <parserStatements> }`

`<transitionStatement> := { transition <stateExpression> }`

`<stateExpression> := <name> ; | <selectExpression>`

`<selectExpression> := select ( <expressionList> ) { <selectCaseList> }`

`<selectCaseList> := { <selectCase> }*`

`<selectCase> := <keysetExpression> : <name> ;`

```
<keysetExpression> := <tupleKeysetExpression> | <simpleKeysetExpression>

<tupleKeysetExpression> := ( <keysetExpressionList> )

<keysetExpressionList> := <simpleKeysetExpression> { , <simpleKeysetExpression }*

<simpleKeysetExpression> := <expression> /* { ( mask | range ) <expression> } */ | default | _

/* <valueSetDeclaration> := valueset '<' ( <baseType> | <tupleType> | <typeName> ) '>'
   '(' <expression> ')' <name> ';' */
```

CONTROL
```
<controlDeclaration> := <controlTypeDeclaration> <optConstructorParameters>
  { <controlLocalDeclarations> apply <blockStatement> }

<controlTypeDeclaration> := control <name> <optTypeParameters> ( <parameterList> )

<controlLocalDeclaration> := <variableDeclaration> | <actionDeclaration> | <tableDeclaration> |
  <instantiation>

<controlLocalDeclarations> := { <controlLocalDeclaration }*
```

EXTERN
```
<externDeclaration> := extern
  ( <nonTypeName> <optTypeParameters> { <methodPrototypes> } ) |
  ( <functionPrototype> ; )

<methodPrototypes> := { <methodPrototype> }*

<functionPrototype> := <typeOrVoid> <name> <optTypeParameters> ( <parameterList> )

<methodPrototype> :=
  ( <functionPrototype> ; ) |
  ( TYPE ( <parameterList> ) ; )   /* constructor */
```

TYPES
```
<typeRef> := <baseType> | <namedType> | <tupleType>

<namedType> := <typeName> | <specializedType> | <headerStackType>

<prefixedType> := { . } TYPE_IDENTIFIER

<tupleType> := tuple < <typeArgumentList> >

<headerStackType> := <typeName> [ <expression> ]

<specializedType> := <typeName> < <typeArgumentList> >

<baseType> :=
  bool | error | string | void |
  int { < <integerTypeSize> > } |
  bit { < <integerTypeSize> > } |
  varbit < <integerTypeSize> >

<integerTypeSize> := INTEGER /* | '(' <expression> ')' */

<typeOrVoid> := <typeRef> | void | IDENTIFIER  /* type variable */

<optTypeParameters> := { < <typeParameterList> > }
```

```
<typeParameterList> := <name> { , <name> }*

<realTypeArg> := _ | <typeRef>

<typeArg> := _ | <typeRef> | <nonTypeName>

<realTypeArgumentList> := <realTypeArg> { , <realTypeArg> }*

<typeArgumentList> := { <typeArg> { , <typeArg> }* }

<typeDeclaration> := <derivedTypeDeclaration> | <typedefDeclaration> | ( <parserTypeDeclaration> ; )
  | ( <controlTypeDeclaration> ; ) | ( <packageTypeDeclaration> ; )

<derivedTypeDeclaration> := <headerTypeDeclaration> | <headerUnionDeclaration> |
  <structTypeDeclaration> | <enumDeclaration>

<headerTypeDeclaration> := header <name> { <structFieldList> }

<headerUnionDeclaration> := header_union <name> { <structFieldList> }

<structTypeDeclaration> := struct <name> { <structFieldList> }

<structFieldList> := { <structField> { , <structField> }* }

<structField> := <typeRef> <name> ;

<enumDeclaration> := enum { bit < INTEGER > } <name> { <specifiedIdentifierList> }

<errorDeclaration> := error { <identifierList> }

<matchKindDeclaration> := match_kind { <identifierList> }

<identifierList> := <name> { , <name> }*

<specifiedIdentifierList> := <specifiedIdentifier> { , <specifiedIdentifier> }*

<specifiedIdentifier> := <name> { = <initializer> }

<typedefDeclaration> := ( typedef | type ) ( <typeRef> | <derivedTypeDeclaration> ) <name> ';'
```

STATEMENTS

```
<assignmentOrMethodCallStatement> := <lvalue>
  ( { < <typeArgumentList> > } ( <argumentList> ) ; ) |
  ( = <expression> ; )

<emptyStatement> := ;

<returnStatement> := return { expression } ;

<exitStatement> := exit ;

<conditionalStatement> := if ( <expression> ) <statement> { else <statement> }

/* To support direct invocation of a control or parser without instantiation. */
<directApplication> := <typeName> . apply ( <argumentList> ) ;

<statement> := <assignmentOrMethodCallStatement> | <directApplication> | <conditionalStatement> |
  <emptyStatement> | <blockStatement> | <exitStatement> | <returnStatement> | <switchStatement>

<blockStatement> := { <statementOrDeclList> }
```

```
<statementOrDeclList> := { <statementOrDeclaration> }*

<switchStatement> := switch ( <expression> ) { <switchCases> }

<switchCases> := { <switchCase> }*

<switchCase> := <switchLabel> : { <blockStatement> }

<switchLabel> := <name> | default

<statementOrDeclaration> := <variableDeclaration> | <statement> | <instantiation>
```

TABLES

```
<tableDeclaration> := table <name> { <tablePropertyList> }

<tablePropertyList> := <tableProperty> { <tableProperty> }*

<tableProperty> :=
  ( key = { <keyElementList> } ) |
  ( actions = { <actionList> } ) |
  ( const entries = { <entriesList> } ) |  /* immutable entries */
  ( { const } <nonTableKwName> = <initializer> ; )

<keyElementList> := { <keyElement> }*

<keyElement> := <expression> : <name> ;

<actionList> := { <actionRef> ; }*

<actionRef> := <prefixedNonTypeName> { ( <argumentList> ) }

<entriesList> := <entry> { <entry> }*

<entry> := <keysetExpression> : <actionRef> ;

<actionDeclaration> := action <name> ( <parameterList> ) <blockStatement>
```

VARIABLES

```
<variableDeclaration> := { const } <typeRef> <name> { = <expression> };
```

EXPRESSIONS

```
<functionDeclaration> := <functionPrototype> <blockStatement>

<argumentList> := { <argument> { , <argument> }* }

<argument> := <expression> | _

/* <kvList> := <kvPair> { ',' <kvPair> }* */

/* <kvPair> := <name> = <expression> */

<expressionList> := { <expression> { , <expression> }* }

<prefixedNonTypeName> := { . } <nonTypeName>

<lvalue> := <prefixedNonTypeName> {
    ( . <name> ) |  /* member selector */
    ( [ <indexExpression> ] )  /* array subscript */
  }*
```

```
<expression> := <expressionPrimary> { <exprOperator> <expression> }*

<expressionPrimary> := <integer> | <boolean> | <string> |
  ( { . } <nonTypeName> ) |
  ( { <expressionList> } ) |   /* <kvList> */
  ( ( <expression> ) ) |
  ( ( ! | ~ | - ) <expression> ) |   /* unary expression */
  ( <namedType> | error ) |   /* member selector, function call */
  ( ( <typeRef> ) <expression> )   /* cast */

<exprOperator> := <binaryOperator> |
  ( . <name> ) |   /* member selector */
  ( [ <indexExpression> ] ) |   /* array subscript */
  ( ( <argumentList> ) ) |   /* function call */
  ( < <realTypeArgumentList> > ) |
  ( = <expression> )   /* <kvPair> */

<integer> := INTEGER

<boolean> := true | false

<string> := STRING

<indexExpression> := <expression> { : <expression> }

<binaryOperator> := * | / | + | - | <= | >= | < | > | != | == | || | && | | | & | << | >>
```

Syntax Tree

PROGRAM

```
<p4program> := <declarationList>decl_list

<declarationList> := { <declaration>[0..n] }*

<declaration> := ( <variableDeclaration> | <externDeclaration> | <actionDeclaration> |
  <functionDeclaration> | <parserDeclaration> | <parserTypeDeclaration> | <controlDeclaration> |
  <controlTypeDeclaration> | <typeDeclaration> | <errorDeclaration> | <matchKindDeclaration> |
  <instantiation> )decl

<name> := STRINGstrname

<parameterList> := { <parameter>[0..n] }*

<parameter> := { in | out | inout }direction <typeRef>type <name>name { <expression> }init_expr

<packageTypeDeclaration> := <name>name { <typeParameterList> }type_params <parameterList>params

<instantiation> := <typeRef>type_ref <argumentList>args <name>name
```

PARSER

```
<parserDeclaration> := <parserTypeDeclaration>proto { <parameterList> }ctor_params
  <parserLocalElements>local_elements <parserStates>states

<parserTypeDeclaration> := <name>name { <typeParameterList> }type_params <parameterList>params

<parserLocalElements> := { <parserLocalElement>[0..n] }*
```

```
<parserLocalElement> := ( <variableDeclaration> | <instantiation> )_element

<parserStates> := { <parserState>_{[0..n]} }+

<parserState> := <name>_name <parserStatements>_{stmt_list} <transitionStatement>_{transition_stmt}

<parserStatements> := { <parserStatement>_{[0..n]>} }*

<parserStatement> := ( <assignmentStatement> | <functionCall> | <directApplication> |
  <parserBlockStatement> | <variableDeclaration> )_stmt

<parserBlockStatement> := <parserStatements>_{stmt_list}

<transitionStatement> := <stateExpression>_stmt

<stateExpression> := ( <name> | <selectExpression> )_expr

<selectExpression> := <expressionList>_{expr_list} <selectCaseList>_{case_list}

<selectCaseList> := { <selectCase>_{[0..n]} }*

<selectCase> := <keysetExpression>_{keyset_expr} <name>_name

<keysetExpression> := ( <tupleKeysetExpression> | <expression> | <defaultKeysetExpression> |
  <dontcareKeysetExpression> )_expr

<tupleKeysetExpression> := <keysetExpressionList>_{expr_list}

<keysetExpressionList> := { <simpleKeysetExpression>_{[0..n]} }+
```

CONTROL

```
<controlDeclaration> := <controlTypeDeclaration>_proto { <parameterList> }_{ctor_params}
  <controlLocalDeclarations>_{local_decls} <blockStatement>_{apply_stmt}

<controlTypeDeclaration> := <name>_name { <typeParameterList> }_{type_params} <parameterList>_params

<controlLocalDeclarations> := { <controlLocalDeclaration>_{[0..n]} }*

<controlLocalDeclaration> := ( <variableDeclaration> | <actionDeclaration> | <tableDeclaration> |
  <instantiation> )_decl
```

EXTERN

```
<externDeclaration> := ( <externType> | <functionPrototype> )_decl

<externType> := <name>_name { <typeParameterList>_{type_params} } <methodPrototypes>_{method_protos}

<methodPrototypes> := { <methodPrototype>_{[0..n]} }*

<functionPrototype> := { <typeRef> }_{return_type} <name>_name { <typeParameterList> }_{type_params}
  <parameterList>_params
```

TYPES

```
<typeRef> := ( <baseTypeBool> | <baseTypeError> | <baseTypeInteger> | <baseTypeBit> | <baseTypeVarbit> |
  <baseTypeString> | <baseTypeVoid> | <namedType> | <tupleType> )_type

<namedType> := ( <name> | <specializedType> | <headerStackType> )_type
```

```
<tupleType> := <typeArgumentList>_{type\_args}

<headerStackType> := <name>_{name} <expression>_{stack\_expr}

<specializedType> := <name>_{name} <typeArgumentList>_{type\_args}

<baseTypeBool> := <name>_{name}
<baseTypeError> := <name>_{name}
<baseTypeInteger> := <name>_{name} { <integerTypeSize> }_{size}
<baseTypeBit> := <name>_{name} { <integerTypeSize> }_{size}
<baseTypeVarbit> := <name>_{name} <integerTypeSize>_{size}
<baseTypeString> := <name>_{name}
<baseTypeVoid> := <name>_{name}

<integerTypeSize> := INTEGER_{size}

<typeParameterList> := { <name>_{[0..n]} }+

<realTypeArg> := ( _ | <typeRef> )_{arg}

<typeArg> := ( _ | <typeRef> | <name> )_{arg}

<realTypeArgumentList> := { <realTypeArg>_{[0..n]} }+

<typeArgumentList> := { <typeArg>_{[0..n]} }*

<typeDeclaration> := ( <derivedTypeDeclaration> | <typedefDeclaration> | <parserTypeDeclaration> |
  <controlTypeDeclaration> | <packageTypeDeclaration> )_{decl}

<derivedTypeDeclaration> := ( <headerTypeDeclaration> | <headerUnionDeclaration> |
  <structTypeDeclaration> | <enumDeclaration> )_{decl}

<headerTypeDeclaration> := <name>_{name} <structFieldList>_{fields}

<headerUnionDeclaration> := <name>_{name} <structFieldList>_{fields}

<structTypeDeclaration> := <name>_{name} <structFieldList>_{fields}

<structFieldList> := { <structField>_{[0..n]} }*

<structField> := <typeRef>_{type} <name>_{name}

<enumDeclaration> := INTEGER_{type\_size} <name>_{name} <specifiedIdentifierList>_{fields}

<errorDeclaration> := <identifierList>_{fields}

<matchKindDeclaration> := <identifierList>_{fields}

<identifierList> := { <name>_{[0..n]} }+

<specifiedIdentifierList> := { <specifiedIdentifier>_{[0..n]} }+

<specifiedIdentifier> := <name>_{name} { <initializer> }_{init\_expr}

<typedefDeclaration> := ( <typeRef> | <derivedTypeDeclaration> )_{type\_ref} <name>_{name}
```

STATEMENTS

`<assignmentStatement> := <lvalueExpression>`$_{\text{lhs\_expr}}$ `<expression>`$_{\text{rhs\_expr}}$

`<functionCall> := ( <expression> | <lvalueExpression> )`$_{\text{lhs\_expr}}$ `<argumentList>`$_{\text{args}}$

`<returnStatement> := { <expression> }`$_{\text{expr}}$

`<exitStatement>`

`<conditionalStatement> := <expression>`$_{\text{cond\_expr}}$ `<statement>`$_{\text{stmt}}$ `<statement>`$_{\text{else\_stmt}}$

`<directApplication> := <typeName>`$_{\text{name}}$ `<argumentList>`$_{\text{args}}$

`<statement> := ( <assignmentStatement> | <functionCall> | <directApplication> | <conditionalStatement> |`
  `<emptyStatement> | <blockStatement> | <exitStatement> | <returnStatement> | <switchStatement> )`$_{\text{stmt}}$

`<blockStatement> := <statementOrDeclList>`$_{\text{stmt\_list}}$

`<statementOrDeclList> := { <statementOrDeclaration>`$_{[0..n]}$ `}*`

`<switchStatement> := <expression>`$_{\text{expr}}$ `<switchCases>`$_{\text{switch\_cases}}$

`<switchCases> := { <switchCase>`$_{[0..n]}$ `}*`

`<switchCase> := <switchLabel>`$_{\text{label}}$ `<blockStatement>`$_{\text{stmt}}$

`<switchLabel> := ( <name> | `**`default`**` )`$_{\text{label}}$

`<statementOrDeclaration> := ( <variableDeclaration> | <statement> | <instantiation> )`$_{\text{stmt}}$

TABLES

`<tableDeclaration> := <name>`$_{\text{name}}$ `<tablePropertyList>`$_{\text{prop\_list}}$

`<tablePropertyList> := { tableProperty`$_{[0..n]}$ `}+`

`<tableProperty> := ( <keyProperty> | <actionsProperty> | <entriesProperty> | <simplePropery> )`$_{\text{prop}}$

`<keyProperty> := <keyElementList>`$_{\text{keyelem\_list}}$

`<keyElementList> := { <keyElement>`$_{[0..n]}$ `}*`

`<keyElement> := <expression>`$_{\text{expr}}$ `<name>`$_{\text{match}}$

`<actionsProperty> := <actionsList>`$_{\text{action\_list}}$

`<actionList> := { <actionRef>`$_{[0..n]}$ `}*`

`<actionRef> := <name>`$_{\text{name}}$ `<argumentList>`$_{\text{args}}$

`<entriesProperty> := <entriesList>`$_{\text{entries\_list}}$

`<entriesList> := { <entry>`$_{[0..n]}$ `}+`

`<entry> := <keysetExpression>`$_{\text{keyset}}$ `<actionRef>`$_{\text{action}}$

`<simplePropery> := <name>`$_{\text{name}}$ `<expression>`$_{\text{init\_expr}}$

```
<actionDeclaration> := <name>name <parameterList>params <blockStatement>stmt
```

VARIABLES

```
<variableDeclaration> := <typeRef>type <name>name { <expression> }init_expr
```

EXPRESSIONS

```
<functionDeclaration> := <functionPrototype>proto <blockStatement>stmt

<argumentList> := { <argument>[0..n] }*

<argument> := ( <expression> | _ )arg

<kvPair> := <name>name <expression>init_expr

<expressionList> := { <expression>[0..n] }*

<lvalueExpression> := ( <name> | <memberSelector> | <arraySubscript> )expr

<expression> := ( <integerLiteral>| <booleanLiteral> | <stringLiteral> | <name> | <namedType> |
  <expressionList> | castExpression | <unaryExpression> | <binaryExpression> |
  <memberSelector> | <arraySubscript> | <functionCall> | <kvPair> )expr { <realTypeArgumentList> }type_args

<integerLiteral> := INTEGERvalue INTEGERwidth

<booleanLiteral> := INTEGERvalue

<stringLiteral> := STRINGvalue

<castExpression> := <typeRef>type <expression>expr

<unaryExpression> := <operator>op <expression>operand

<binaryExpression> := <expression>left_operand <operator>op <expression>right_operand

<memberSelector> := <expression>lhs_expr <name>name

<arraySubscript> := <expression>lhs_expr <indexExpression>index_expr

<indexExpression> := <expression>start_index { <expression> }end_index

<operator> := ✕ | / | + | - | ≤ | ≥ | < | > | ≠ | = | | | & | ~ | ⋁ | ⋀ | ⌐ | << | >>
```